

PI 1093500

REC'D 21 NOV 2003

WIPO PCT



# THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE  
United States Patent and Trademark Office

November 14, 2003

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM  
THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK  
OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT  
APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A  
FILING DATE.

APPLICATION NUMBER: 60/407,929

FILING DATE: September 04, 2002

RELATED PCT APPLICATION NUMBER: PCT/US03/27693



By Authority of the  
COMMISSIONER OF PATENTS AND TRADEMARKS

  
M. SIAS  
Certifying Officer

**PRIORITY  
DOCUMENT**

SUBMITTED OR TRANSMITTED IN  
COMPLIANCE WITH RULE 17.1(a) OR (b)

BEST AVAILABLE COPY

This collection of information is required by 37 CFR 1.51. The information is used by the public to file (and by the PTO to process) a provisional application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the complete provisional application to the PTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Washington, D.C. 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C. 20231.

Approved for use through 10/31/2002. OMB 0651-0032  
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number

# FEE TRANSMITTAL for FY 2002

Patent fees are subject to annual revision

☒ Applicant claims small entity status. See 37 CFR 1.27

TOTAL AMOUNT OF PAYMENT

(\$)80.00

## Complete if Known

Application Number

Filing Date

First Named Inventor

Examiner Name

Group Art Unit

Attorney Docket No.

## METHOD OF PAYMENT (check all that apply)

☒ Check ☐ Credit card ☐ Money Order ☐ Other ☐ None

☐ Deposit Account

Deposit Account Number  
Deposit Account Name

The Commissioner is authorized to: (check all that apply)

☐ Charge fee(s) indicated below ☐ Credit any overpayments

☐ Charge any additional fee(s) during the pendency of this application

☐ Charge fee(s) indicated below, except for the filing fee to the above-identified deposit account

## FEE CALCULATION

### 1. BASIC FILING FEE

Large Entity | Small Entity

Fee Code (\$)	Fee Code (\$)	Fee Description	Fee Paid
101 740	201 370	Utility filing fee	
106 330	206 165	Design filing fee	
107 510	207 255	Plant filing fee	
108 740	208 370	Reissue filing fee	
114 160	214 80	Provisional filing fee	<u>80</u>

SUBTOTAL (1) (\$)80.00

### 2. EXTRA CLAIM FEES FOR UTILITY AND REISSUE

Total Claims	Extra Claims	Fee from below	Fee Paid
Independent Claims	-20** =	X	
Multiple Dependent	-3** =	X	

Large Entity | Small Entity

Fee Code (\$)	Fee Code (\$)	Fee Description
103 18	203 9	Claims in excess of 20
102 84	202 42	Independent claims in excess of 3
104 280	204 140	Multiple dependent claim, if not paid
109 84	209 42	** Reissue independent claims over original patent
110 18	210 9	** Reissue claims in excess of 20 and over original patent

SUBTOTAL (2) (\$)

\*\*or number previously paid, if greater; For Reissues, see above

## FEE CALCULATION (continued)

### 3. ADDITIONAL FEES

Large Entity | Small Entity

Fee Code (\$)	Fee Code (\$)	Fee Description	Fee Paid
105 130	205 65	Surcharge - late filing fee or oath	
127 50	227 25	Surcharge - late provisional filing fee or cover sheet	
139 130	139 130	Non-English specification	
147 2,520	147 2,520	For filing a request for <i>ex parte</i> reexamination	
112 920*	112 920*	Requesting publication of SIR prior to Examiner action	
113 1,840*	113 1,840*	Requesting publication of SIR after Examiner action	
115 110	215 55	Extension for reply within first month	
116 400	216 200	Extension for reply within second month	
117 920	217 460	Extension for reply within third month	
118 1,440	218 720	Extension for reply within fourth month	
128 1,960	228 980	Extension for reply within fifth month	
119 320	219 160	Notice of Appeal	
120 320	220 160	Filing a brief in support of an appeal	
121 280	221 140	Request for oral hearing	
138 1,510	138 1,510	Petition to institute a public use proceeding	
140 110	240 55	Petition to revive - unavoidable	
141 1,280	241 640	Petition to revive - unintentional	
142 1,280	242 640	Utility Issue fee (or reissue)	
143 460	243 230	Design Issue fee	
144 620	244 310	Plant issue fee	
122 130	122 130	Petitions to the Commissioner	
123 50	123 50	Processing fee under 37 CFR 1.17(q)	
126 180	126 180	Submission of Information Disclosure Stmt	
581 40	581 40	Recording each patent assignment per property (times number of properties)	
146 740	246 370	Filing a submission after final rejection (37 CFR § 1.129(a))	
149 740	249 370	For each additional invention to be examined (37 CFR § 1.129(b))	
179 740	279 370	Request for Continued Examination (RCE)	
169 900	169 900	Request for expedited examination of a design application	

Other fee (specify)

\*Reduced by Basic Filing Fee Paid

SUBTOTAL (3) (\$)

SUBMITTED BY

Fraunhofer CREG, Inc.

Complete (if applicable)

Name (Print/Type)

Chenghui Luo

Registration No.  
(Attorney/Agent)

Telephone

Signature

*Chenghui Luo*

Date

8/30/02

**WARNING:** Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO2038.

Burden Hour Statement: This form is estimated to take 0.2 hours to complete. Time will vary depending upon the needs of the individual case. Any comments on the amount of time you are required to complete this form should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Assistant Commissioner for Patents, Washington, DC 20231.



**Fraunhofer** Center for Research in  
Computer Graphics, Inc.

321 South Main Street  
Providence, RI 02903 USA

Phone +1 401 453 6363  
Fax +1 401 453 0444  
<http://www.crcg.edu>

Assistant Commissioner for Patents  
Box PPA  
Washington, DC 20231

August 30, 2002

Dear Sir / Madam,

Enclosed please find a Provisional Patent Application entitled "Multi-layer Protection of Mobile Code". Please find enclosed in the submission:

- This cover letter.
- The provisional application for patent cover sheet.
- 18 pages of specifications.
- ~~A stamped self-addressed postcard.~~
- Fee transmittal of \$80 for the Small Entity PPA Filing Fee.

**We are claiming a Small Entity reduction of the transmittal fee as Fraunhofer CRCG is a non-profit organization.**

~~Please mail the postcard on receipt of this letter.~~ Thank you!

Yours Faithfully,

Chenghui Luo, Ph.D.



The International Network of Institutions  
for advanced education, training and R&D in Computer Graphics technology, systems and applications

# **MULTI-LAYER PROTECTION OF MOBILE CODE**

**Provisional Patent Application**

**Chenghui Luo, Ph.D.**

*Secure Distributed Technologies  
Fraunhofer Center for Research in Computer Graphics, Inc.  
321 South Main St.  
Providence, RI 02903*

Phone: (+1) 401-453-6363 x208  
FAX: (+1) 401-453-0444  
EMAIL: cluo@crcg.edu  
URL: <http://www.crcg.edu>

*Aug. 30, 2002*

## Executive Summary

Mobile code is computer code that is produced on one node of a computer network but is transferred and executed on another network node. A mobile agent is a mobile software module composed of autonomous mobile code. Mobile agent software is a promising new computing paradigm for distributed computing in a large computer network environment, because of the autonomy and efficiency of mobile code.

However, mobile code is vulnerable to security problems, including malicious agents, and malicious hosts. A malicious agent is an agent that performs harmful actions on an agent host. Possible damaging actions include unauthorized access, modification and overuse of local resources, such as sensitive data, system calls, and CPU time. A malicious host is a host computer that performs harmful actions on a mobile agent. Because an agent is executed in an environment that an agent host provides, there are a number of ways that an agent host can attack an agent, including spying out and manipulating agent code, data, and control flow; listening to and tampering with data exchange between an agent and agent owner; executing code incorrectly; and denying execution and masquerading as another host. In addition, reverse engineering (the process of decompilation of a mobile agent bytecode to obtain its source code) is also a great threat for Java-based mobile agents. That is, if a malicious host can obtain the source code of an agent, the agent can be manipulated with possibly harmful results.

All of the information attacks listed above, for example, become much easier based on the analysis of the source code of a mobile agent. To defend against terrorist attacks in cyberspace, it is clear that mobile code — as part of the nation's information infrastructure — must be well-protected.

In the course of the Mobile Code Security (MCS) project supported by the US Air Force Research Laboratory, a MCS Framework has been developed by Fraunhofer CRCG. The MCS Framework provides a multi-layered approach to safeguarding mobile agents, and includes technologies to support

- *complete* obfuscation,
- encrypted execution, and
- code watermarking technologies.

### Complete Obfuscation

Code obfuscation is a technology to hide or remove symbolic information in mobile code so that source code can be protected even if a program is decompiled. Using the MCS Framework developed by Fraunhofer CRCG, mobile code is protected up to the Java system class level. This technology provides mobile code complete obfuscation from application classes to system classes, so it is a much stronger protection than traditional obfuscation approaches which only obfuscate application classes. Mobile code protected with the MCS Framework is very difficult to reverse-engineer and manipulate, and this technique can also be used for protection of intellectual property rights and classified data.

## **Encrypted Execution**

An encrypted execution framework based on a customized Java class loader has also been developed as part of the MCS Framework. With the customized mobile code class loader, Java system classes remain obfuscated and encrypted at run time, therefore static or dynamic analysis of the mobile code is extremely unlikely, which greatly enhances mobile code security.

## **Code Watermarking**

Code watermarking is a technology to embed a secure and invisible label in mobile code. Watermarks embedded in mobile code can authenticate not only mobile code, but watermarks embedded in system classes can be used to authenticate the execution environment on a remote host. The robustness of a watermark measures the difficulty of removing the watermark from mobile code. A non-robust watermark is called a "fragile watermark," and it is this fragility that helps to authenticate mobile code. More robust watermarks can survive piracy attacks, and thus can be used for intellectual property protection. The fragile Java bytecode watermarking techniques developed at Fraunhofer CRCG currently support mobile code authentication. Robust watermarking techniques for copyright protection have been designed and are currently being implemented. The encrypted execution mechanism described above also supports secure execution of watermarked mobile code, which provides additional protection from attack for software watermarks.

Remote execution of mobile code was formerly risky and dangerous in an insecure malicious host environment. However, using the MCS Framework developed at Fraunhofer CRCG — consisting of complete obfuscation, encrypted execution and code watermarking — mobile code can be safely executed on remote hosts. With security risks minimized, large-scale mobile agent-based applications such as network management can now be developed.

## TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION AND OVERVIEW .....</b>	<b>5</b>
1.1	THE MOBILE CODE SECURITY PROBLEM .....	5
1.2	SOLUTIONS DEVELOPED BY FRAUNHOFER CRCG .....	5
<b>2</b>	<b>RESEARCH BACKGROUND .....</b>	<b>6</b>
2.1	MOBILE CODE SECURITY PROBLEMS .....	7
2.2	RELATED WORK .....	7
2.2.1	<i>Secure Hardware</i> .....	8
2.2.2	<i>Mobile Cryptography</i> .....	8
2.2.3	<i>Time Limited Blackbox</i> .....	8
2.2.4	<i>Detection Objects</i> .....	9
2.2.5	<i>Protective Assertions</i> .....	9
2.2.6	<i>Cryptographic Traces</i> .....	9
<b>3</b>	<b>FRAUNHOFER CRCG MOBILE CODE SECURITY FRAMEWORK .....</b>	<b>9</b>
3.1	THE FRAUNHOFER CRCG 3-LAYER SOLUTION .....	9
3.2	COMPLETE OBFUSCATION .....	11
3.3	ENCRYPTED EXECUTION .....	13
3.4	CODE WATERMARKING .....	15
<b>4</b>	<b>CONCLUSIONS .....</b>	<b>16</b>
<b>5</b>	<b>REFERENCES .....</b>	<b>17</b>



# MULTI-LAYER PROTECTION OF MOBILE CODE

Chenghui Luo, Ph.D.

*Secure Distributed Technologies  
Fraunhofer Center for Research in Computer Graphics, Inc.  
321 South Main St.  
Providence, RI 02903*

## 1 Introduction and Overview

Mobile code is computer code that is produced on one node of a computer network but is transferred and executed on another network node. A mobile agent is a mobile software module composed of autonomous mobile code. Mobile agent software is a promising new computing paradigm for distributed computing in a large computer network environment, because of the autonomous and efficient nature of mobile code.

### 1.1 The Mobile Code Security Problem

However, the mobile code is vulnerable to security problems, including malicious agents, and malicious hosts. A malicious agent is an agent that performs harmful actions on an agent host. Possible damaging actions include unauthorized access, modification and overuse of local resources, such as sensitive data, system calls, and CPU time. A malicious host is a host computer that performs harmful actions on a mobile agent. Because an agent is executed in an environment that an agent host provides, there are a number of ways that an agent host can attack an agent, including spying out and manipulating agent code, data, and control flow; listening to and tampering with data exchange between an agent and agent owner; executing code incorrectly; and denying execution and masquerading as another host.

In addition, reverse engineering (the process of decompilation of a mobile agent bytecode to obtain its source code) is also a great threat for Java-based mobile agents. That is, if a malicious host can obtain the source code of an agent, the agent can be manipulated with possible harmful results.

All of the information attacks listed above become much easier based on the analysis of the source code of a mobile agent. To defend against terrorist attacks in cyberspace, it is clear that mobile code — as part of the nation's information infrastructure — must be well-protected.

### 1.2 Solutions Developed by Fraunhofer CRCG

Fraunhofer CRCG's solution to the mobile code security problems is its Mobile Code Security Framework, including complete obfuscation, encrypted execution and code watermarking technologies. These technologies were developed in 2001 with the support of the US Air Force Research Laboratory.

### **Complete Obfuscation**

Code obfuscation is a technology to hide or remove symbolic information in mobile code so that source code can be protected even if a program is decompiled. Using the MCS Framework developed by Fraunhofer CRCG, mobile code is protected up to the Java system class level. This technology provides mobile code complete obfuscation from application classes to system classes, so it is a much stronger protection than traditional obfuscation approaches which only obfuscate application classes. Mobile code protected with the MCS Framework is very difficult to reverse-engineer and manipulate, and this technique can also be used for protection of intellectual property rights and classified data.

### **Encrypted Execution**

An encrypted execution framework based on a customized Java class loader has also been developed as part of the MCS Framework. With the customized mobile code class loader, Java system classes remain obfuscated and encrypted at run time, therefore static or dynamic analysis of the mobile code is extremely unlikely, which greatly enhances mobile code security.

### **Code Watermarking**

Code watermarking is a technology to embed a secure and invisible label in mobile code. Watermarks embedded in mobile code can authenticate not only mobile code, but watermarks embedded in system classes can be used to authenticate the execution environment on a remote host. The robustness of a watermark measures the difficulty of removing the watermark from mobile code. A non-robust watermark is called a "fragile watermark," and it is this fragility that helps to authenticate mobile code. More robust watermarks can survive piracy attacks, and thus can be used for intellectual property protection. The fragile Java bytecode watermarking techniques developed at Fraunhofer CRCG currently support mobile code authentication. Robust watermarking techniques for copyright protection have been designed and are currently being implemented. The encrypted execution mechanism described above also supports secure execution of watermarked mobile code, which provides additional protection from attack for software watermarks.

Remote execution of mobile code was formerly risky and dangerous in an insecure malicious host environment. However, using the MCS Framework developed at Fraunhofer CRCG — consisting of complete obfuscation, encrypted execution and code watermarking — mobile code can be safely executed on remote hosts. With security risks minimized, large scale mobile agent-based applications such as network management can now be developed.

## **2 Research Background**

Mobile code is now ubiquitous on the Internet. Many web pages become "active" by including mobile code such as Java™ applets and JavaScript or ActiveX scripts. Mobile code is also used to implement features in devices such as cellular phones. When a user accesses one of these features on a cellular phone, mobile code for the feature is downloaded to the cellular phone and then used in the interactions that involve the feature.

When mobile code becomes an autonomous program and travels from host to host on a network, it evolves into *mobile agents*. Compared to mobile code, mobile agents typically move from host to host to accomplish specified missions autonomously and collaboratively. Mobile agents emerge as promising distributed computing paradigm to replace the conventional client-server model. With mobile agent platforms such as IBM Aglets [Karjoth et al, 1998] and Mitsubishi Concordia [Walsh et al, 1998], more and more mobile agent applications such as travel agents, auction agents and e-commerce agents, can be conveniently developed. Partially supported by DARPA and ARL during 1997-1999, Fraunhofer CRCG also developed a Digital Watermark Agent application with CRCG's own lightweight mobile agent platform [Zhao and Luo, 1999].

## 2.1 Mobile Code Security Problems

Because mobile code originates from a remote, possibly "untrusted" system, mobile code could be a potential security threat to a mobile agent host, e.g., unauthorized access, modification and overuse of local resources, such as sensitive data, system calls, and CPU time, just like viruses. This problem is called "malicious agent" problem and although this problem seems the major and natural security problem for a mobile agent host, it can be basically solved by confining a mobile agent in a sandbox with fine-grained access control policies. A surprisingly much more difficult problem is the "malicious host", which can exploit a mobile agent almost freely, because a mobile agent is executed in an environment that is provided and controlled by the malicious host, and thus can be easily accessed by a mobile agent host. Because an agent is executed in an environment that an agent host provides, there are a number of ways that an agent host can attack an agent [Hohl, 1998], including

- *spying out* and manipulating agent code, data, and control flow;
- listening to and tampering with data exchange between an agent and agent owner;
- executing code incorrectly; and
- denying execution and masquerading as another host.

Besides these malicious host attacks, agent reverse engineering, the process of decompilation of a mobile agent bytecode to obtain its source code, is also a great threat for mobile agents [Luo and Zhao, 1999]. Today, it is not a secret that Java programs (applets or applications) can be easily decompiled and reverse engineered from Java bytecode to Java source code. Although all computer programs are theoretically subject to decompilation, Java's rich symbolic information in class files and Java's dynamic link mechanism makes the chance of successfully decompiling Java programs great, using some commercial decompilers. The consequence of decompilation is serious, e.g., it will become cheap and easy to steal intellectual property or classified data embedded in mobile code, to alter legitimate programs and redistribute them, and for the case of a mobile agent, a malicious host can easily manipulate a mobile agent. All of the information attacks listed above, for example, become much easier based on the analysis of a mobile agent's source code. Because of these reasons, research efforts of Fraunhofer CRCG have focused on the protection of mobile code against malicious hosts.

## 2.2 Related Work

The malicious host problem is recognized by many researchers as a much harder task;

currently there are many diverse opinions about this problem and various solutions have been proposed to solve it.

### **2.2.1 Secure Hardware**

Some researchers argued that it is doubtful that an agent could keep a secret (e.g., a secret key for decryption) [Gong 1997], since the information belonging to a mobile agent is completely available to its host system. Some even claimed that it is impossible to prevent agent tampering unless trusted hardware is available in agent platforms [Karjoth et al. 1998]. Such belief gives an impression that the hardware approach is the only one that can lead to highly secure agent systems. Currently there are some commercial products in this category, such as smart cards, including Sun Microsystems' Java Card [Sun Microsystems 1997]. However, a closer look into Java Card finds that currently Java Card is not sufficient for large-scale computation. For example, the current specification of the Java Card only supports a small subset of the Java language, and it is not able to load classes dynamically, though it is enough for typical smart card applications, such as payment and some intelligent features. A hardware approach could be successful, provided that serious hardware can be developed and somehow integrated into current computer hardware. Otherwise, it is not a practical approach.

### **2.2.2 Mobile Cryptography**

Although protecting mobile agents against tampering attempts by the executing host is much harder, not everyone believes that computation privacy for mobile code is impossible without tamper resistant hardware, or that an agent cannot keep a secret, such as a secret key. In [Sander and Tschudin 1998], for the first time, it was pointed out that indeed there was an error in this reasoning. They explained that although that opinion was true for agents in clear text form, it was not for agents in cryptographic form in which both the data and the functions of an agent were encrypted. In such a case, through computing with encrypted data and function, a software-only approach for providing computation privacy for mobile code is possible. Sander and Tschudin demonstrated this approach with a mobile code fragment that computes an algebraic circuit (a polynomial). They also described another method in which a mobile agent could digitally sign its output securely. This second approach is on the right track, and with modification and customization, the basic idea of computing with encrypted data and function could work successfully for more and more mobile agent applications. But in practice, this approach is still incomplete, because the transformation that Sander and Tschudin used can only be applied to algebraic circuits, a universal transformation is hard to construct.

### **2.2.3 Time Limited Blackbox**

In [Hohl 1998], another approach was presented that partially solved the malicious host problem. The idea here is to create a blackbox out of an original agent. A blackbox is an agent that performs the same work as the original agent, but is of a different structure. This difference allows the agent dispatcher to assume a certain agent protection time interval, during which it is impossible for an attacker to discover relevant data or manipulate the execution of the agent. After that time interval the agent and some associated data become invalid, and the agent cannot migrate or interact anymore, which prevents the exploitation of attacks after the protection interval. This idea seems vague because it is hard to implement with different CPU clocks and

execution environments, and there are no mechanisms to block a host's access to an agent's secret data.

#### 2.2.4 Detection Objects

Because complete prevention of a host's attacks on an agent is hard to achieve, some researchers have turned to methods that detect evidence of a host's attacks. In [Meadows 1997], a rough idea about "detection objects" was proposed. Detection objects are objects sent to hosts solely for detection purposes, therefore they may contain some data or function to fool a host system so that if an host tries to tamper with them, the objects dispatcher has a way to detect the host's misbehavior. This way, an agent dispatcher can verify if a host is a malicious one.

#### 2.2.5 Protective Assertions

A similar approach, called "protective assertions", was presented in [Kassab & Voas 1998]. Their goal was to make it much harder for an agent to be tampered with. By inserting some assertions into the agent source code, an agent can check the conditions specified in the assertions at runtime. Agent assertions are usually Boolean functions that evaluate to be true when an agent execution state satisfies a semantic condition, and false otherwise. Kassab et al. also proposed to let the agent assertions output more than Boolean values in order to increase the agent observability. Based on the output results of assertion checking which are sent to the agent dispatcher, an agent dispatcher (owner) has a way to detect if its agent has been attacked.

#### 2.2.6 Cryptographic Traces

In [Vigna 1998], a detection mechanism was presented based on execution cryptographic traces. It allows one to detect attacks against code, state, and the execution flow of mobile software components. Cryptographic traces are logs of the operations performed by an agent during its lifetime. Using these traces, an agent dispatcher can check — following agent termination — to see if the execution history of the agent conforms to a correct execution. For example, an agent dispatcher may verify part of the agent's logs, to discover if there are any signs of a host's attacks, by repeating the agent's execution on the same secure environment. This method gives a somewhat reasonable way for the detection of malicious hosts, but how to generate this cryptographic trace is not further discussed.

Based on this research review, although there was much research work towards the malicious host problem, the problem is far from being solved. Based on the current research, and previous work at Fraunhofer CRCG on mobile agent security, it was found that a multi-layer approach is practical — in this case, a mobile agent could be armed with 3 layers of *information armors* as described below.

### 3 Fraunhofer CRCG Mobile Code Security Framework

#### 3.1 The Fraunhofer CRCG 3-layer Solution

The *malicious host* is a very difficult problem, and some researchers have even claimed that it is impossible to verify if an agent is correctly executed on a malicious host [Karjoth et al.,

1998]. After reviewing the literature, it was determined that a multi-layer approach could provide mobile agents much better protection. Please refer to Figure 1. The mobile agent protection mechanism developed at Fraunhofer CRCG consists of 3 layers of *information armor*:

1. The first layer is agent code obfuscation. This layer significantly hides symbolic information such as class, field and method names; prevents understanding of an agent's source code based on static analysis; and helps to protect an agent's internal procedures and secrets.
2. The second layer is for secure execution of completely obfuscated agent code. This layer provides a customized class loader to dynamically load obfuscated mobile code and execute the code in an encrypted fashion. Both the first (obfuscation) and the third (dynamic watermarking) layers are supported through this layer.
3. The third layer is to watermark an agent's code, which is used to detect attacks from a malicious host. This third layer allows a mobile agent to detect intrusion and dispatch information back to its agency (original agent dispatcher), so that the agency is notified that its agent has been attacked.

The integration of all of these complementary layers greatly limits the possibility of successful malicious attacks. In previous research, some techniques such as secret spreading, assertions [Kassab & Voas 1998] and time constraints [Hohl 1998], were quite different from each other, and it has been difficult to integrate them together into a single, mobile agent application. Based on CRCG's customized class loader-based encrypted execution framework, these three layers of *information armor* have been successfully integrated into Fraunhofer CRCG mobile code security framework and it supports secret spreading and assertions in a consistent way. This greatly simplifies the complexity of mobile code security management.

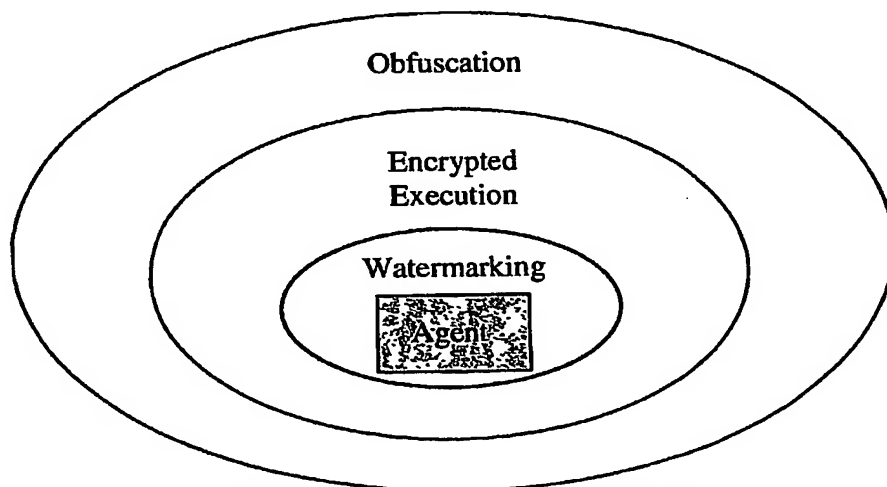


Figure 1. CRCG 3-layer information armor for agent protection.

## 3.2 Complete Obfuscation

There exist a few obfuscation techniques [Collberg, 1997]. For example, data obfuscation changes the storage modes, encoding methods, aggregation, and/or ordering of data structures. Control obfuscation may hide the real control flow behind irrelevant or never executed statements, or change the reducible code to non-reducible. All of these obfuscators generally take the common approach of renaming meaningful symbolic names to meaningless names. This technique effectively renders the decompiled source code difficult for a human being to understand, even if one has plain text source code. Some of these obfuscators also obfuscate the control flow of a Java program, e.g., by inserting many GoTo instructions to confuse decompilers. In practice, this approach works well, and not many decompilers can correctly analyze the obfuscated control flow and output the corresponding source code.

As seen above, one common way of obfuscating a program is to replace all of the names in the program with names that are legal in the programming language but are as meaningless as possible to a human being reading the program. One of the key observations in the Fraunhofer CRCG research effort has been that this kind of technique is generally only applied to an bytecode of an application package, but not to the Java library system classes. Without a developer's meaningful symbolic names, an obfuscated package is harder to understand, but all the meaningful symbolic names of Java library classes in source code still gives a clear clue about the internal logic of a Java package.

Directly extending traditional obfuscation techniques to Java library classes will simply make an application package not executable, because a Java virtual machine may not be able to link those application and library classes. Although Java's design essentially make it vulnerable to decompilation, it is also Java's dynamic class loading feature that make it possible to fix the hole. Simply speaking, the *complete obfuscation* technique developed at Fraunhofer CRCG obfuscates *all* symbolic information in a Java software package including static data, and uses a customized Java class loader to load, link and execute the completely obfuscated package.

The Fraunhofer CRCG obfuscation framework contains the following three major components:

- symbolic name obfuscating,
- library-related symbolic name encrypting, and
- runtime library-related symbolic name recovering.

The encrypting and recovering steps are associated with a key.

At the symbolic name obfuscation step, all symbolic information are transformed to meaningless names, for both the application and library level classes. Next, for security reasons, to prevent attackers from finding the original library classes, all library-related symbolic information in an application is encrypted using a key. Finally, at runtime, the symbolic information of library classes will be recovered by encrypting Java library classes with the same encryption key so that the two

encrypted symbolic names can link and execute. This global picture is illustrated in the following figure.

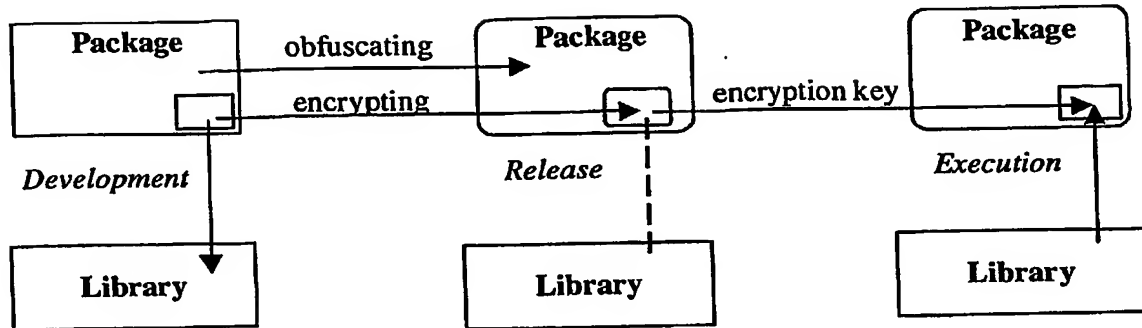


Figure 2. The Complete Obfuscation Framework

The above three components correspond to three different time periods of the software: the obfuscating and encrypting processes happen after the development and before the release of the software; and after a software is released, the recovering process happens at the software execution phase. At the development and the execution phases, the software package is well-linked with a library, but during the release and before the execution phase, the library data names remain encrypted and the package is not well linked with the library.

Library data linking is a very critical component in this obfuscation architecture. For security reasons, this part must be made hard enough for a pirate to attack at runtime. To achieve this goal, in the Fraunhofer CRCG framework, the plain library data names are first encrypted with a key, and then the encrypted names are mapped to the obfuscated library data names. Thus, in the release version of a software package, the library data names are both encrypted and obfuscated.

To link and execute the package, one must have a key to encrypt library-related symbolic names. Note that here one does not use a key to decrypt those names because doing so will reveal the original library names. Instead, one encrypts the library classes with the original encryption key. See Figure 3. This way, the library classes can still be linked together.

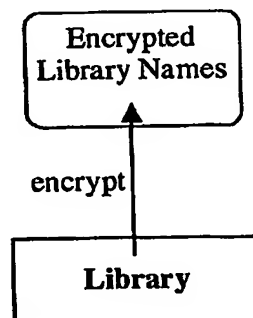


Figure 3. Encryption to link



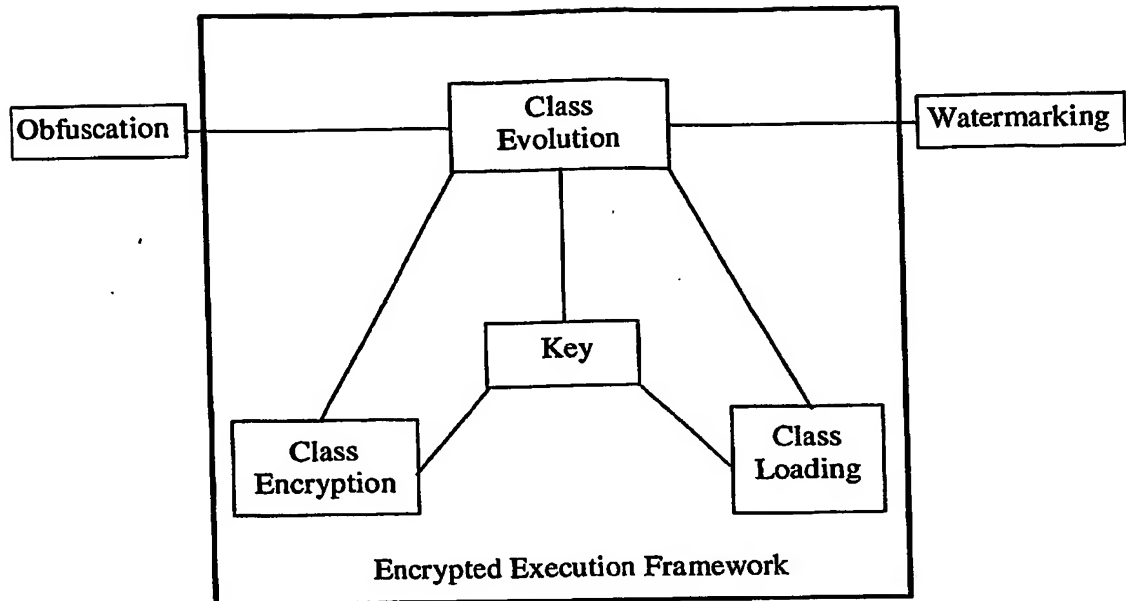
### 3.3 Encrypted Execution

Encrypted execution (also called “computing with encrypted data” or “computing with encrypted function”) might be the last resort for strong mobile code protection. This approach is a pure software approach, and its key point is to execute an agent on a host while the agent keeps its data (or function) encrypted [Sander and Tschudin, 1998]. But for computing with encrypted data, a large set of natural privacy homomorphism approaches are shown to be insecure [Brickell and Odlyzko, 1992], and a secure privacy homomorphism is hard to construct (for example, some approaches only work for polynomials [Sander and Tschudin, 1998]), which renders this encrypted data approach impractical for mobile code protection.

The major drawback of computing with encrypted function is complexity in terms of computation, communication and rounds, and this is the cost of having very strong security properties related to cryptographic or information-theoretic assumptions. The complexity and interactive rounds of the encrypted function approach is also not practical for mobile code, since it makes it very difficult to autonomously deploy and execute tasks remotely.

Based on the above review, it was found by Fraunhofer CRCG that although the encrypted computing approach was on the right track, many open research questions remained, such as how to encrypt data and functions to achieve the non-interactivity of mobile computing. For practical strong security, a class loader-based encrypted execution framework was developed by Fraunhofer CRCG.

The idea of the Fraunhofer CRCG class loader-based encrypted execution framework is to execute a mobile code package with encrypted Java system classes, through the means of the Fraunhofer CRCG customized class loaders. Please refer to Figure 4. This way, although the whole mobile code package is not completely encrypted during execution, if the system classes are encrypted, static or dynamic analysis of the mobile code is much harder, because all the system-related parts are encrypted in the mobile code application.



**Figure 4. Customized Class Loader-based Encrypted Execution Framework**

To make mobile code even harder to modify at both static and dynamic code level, a *class evolution* solution has been developed by Fraunhofer CRCG. Class evolution means that the definition of a class can be changed at runtime through a customized class loader. Even with a debugger, a class controlled in this way is hard to modify, because certain bytecode or internal states of a mobile code application may remain encrypted, and an attacker will not be sure which code or data to modify at a certain time during the execution. Class evolution is also used for the dynamic watermarking techniques developed by Fraunhofer CRCG (section 3.4).

Compared with the general purpose encrypted computing approach, the class loader-based encrypted execution framework developed by Fraunhofer CRCG may be considered a software engineering-oriented approach because the encryption and execution occurs within customized Java class loaders, instead of keeping all the data and functions encrypted during execution as in the case of encrypted computing.

Recall that decompilation is one of the major practical means of malicious host attacks. This class loader-based encrypted execution framework significantly deters this attack, because with partial encryption, all the classes are correctly generated and linked based on the encrypted system classes only at runtime, thus source code is hard to obtain through decompilation of static bytecode of a mobile code application. This shift of static information to runtime generation is very useful for preventing decompilations.

This customized class loader-based encrypted execution framework is well adapted to the Fraunhofer CRCG 3-layer information armor for mobile code security. This framework is seamlessly integrated with

- the obfuscation framework that protects the source code of mobile code,

- the watermarking component which supports detection of malicious host attacks,
- dynamic loading and class evolution for assertions and time limited constraints.

### 3.4 Code Watermarking

Generally speaking, a watermark is an invisible, robust, and secure label embedded in a digital content for the purpose of ownership identification. Software watermarks are used for software copyright protection, just same as the media watermark case [Cox et al, 1996; Zhao and Koch, 1995]. In a mobile code scenario, however, watermarks embedded in mobile code can be used to detect attacks from mobile hosts.

The idea of watermarking agent code is as follows. Before an agent is dispatched, it is first watermarked by the agency. When the watermarked agent arrives at a host, it signals its arrival and executes. Upon completion, the agent will check its watermark, and communicate back to the remote agency. If the watermark has been changed or cannot be found, then the agent will detect that, and notify the remote agency. If, however, the agency does not receive a signal at all from an agent within some expected time interval after arrival at a host, the agency knows that the agent has been significantly modified. So an agency can use this as a way to detect modification of agent code by a malicious host.

There are currently two approaches to watermark a Java program: to embed a watermark *statically* in the program's bytecode, or *dynamically* in the status of a program's execution. Generally, for all watermarking techniques — both media and software watermarking — *robustness* is a major problem. But for *software* watermarking, the *invisibility* requirement is an additional difficulty. Research by Fraunhofer CRCG has shown that many of these watermarking techniques require some amount of extra code in a certain format to be inserted into a program, such as [Monden et al, 2000], [Townsend and Collberg, 2000] and [Palsberg et al, 2000]. Some of them even require that the insertion occur at the source code level, which complicates software engineering and maintenance.

The static watermark embedded in mobile code developed by Fraunhofer CRCG is a simple and easy scheme. Given a key, one first generates a set of locations in the bytecode (based on also some class information, such as the number of bytecodes in a method. This issue is discussed in section 4.3). Based on these locations, one inserts a bytecode “nop” — a no-op instruction which does nothing in a Java virtual machine, and thus doesn't change the functionality of the mobile code. When this same operation is performed at some later time to retrieve the watermark on a bytecode stream, the watermark is valid if and only if the result of this secret operation is identical to the watermark value.

Compared with other static watermark techniques, the Fraunhofer CRCG implementation does not rely on a Java application's source code, but *only* bytecode. This approach relieves software developers from the burden of maintaining watermark-related source code.

Fraunhofer CRCG uses *class evolution* to construct a dynamic watermark. First one inserts into the class a few extra, secret assertion methods with the known watermark as return value under certain conditions. Then these assertion methods are encrypted by encrypting blocks

of the bytecode of the methods with the given watermark key. Other class methods are encrypted by scrambling based on other keys or runtime values of certain fields. Next, each class is encrypted using class evolution (class evolution is a runtime class transforming mechanism that was used for encrypted execution. Here it is used for watermarking purposes). To do class evolution, one can develop a class loader for this class and make it dynamically load the bytecode streams of all of its methods from a byte array generated from all the methods, including the extra methods.

To retrieve the watermark, execute the watermarked class using the customized class loader. Load the regular methods, then with the given watermark retrieval key, decrypt the bytecode of the extra methods and then load and execute them. The original watermark will be returned if there are no modifications.

This dynamic watermarking technique can detect illegal runtime modifications, because if the assertions fail, the return value will be different from the original one, and the mobile code owner will know that this code has been executed incorrectly. Since the encrypted assertion method will be executed as usual, *without* giving extra clues that the agent is detecting modifications, this detection module may not be visible to a malicious host.

## 4 Conclusions

While mobile code-based computing is a promising new computing paradigm, security is a bottleneck, especially because it is very difficult to protect a mobile agent against malicious host attacks, such as decompilation. To better protect a mobile agent, various obfuscation techniques have been proposed and developed. In this research, Fraunhofer CRCG developed innovative *complete* obfuscation technologies, which push the frontier of obfuscation to Java library classes, and Fraunhofer CRCG further achieved stronger security by obfuscating and encrypting the Java library class-related symbolic names in a Java package.

Research on general mobile code security has shown that secure execution may be the utmost solution of mobile code security problem. To achieve stronger mobile code security, Fraunhofer CRCG designed and developed a Java system library based encrypted execution framework. This framework makes use of Java's sophisticated class loading mechanism to load Java system classes and in execution of mobile code, it keeps these system classes hidden and encrypted. This approach naturally integrates with the Fraunhofer CRCG *complete obfuscation* framework, and it significantly improves the security of mobile code. Static analysis and dynamic modification of the mobile code are now much harder within this framework.

While encrypted execution makes it almost impossible for a malicious host to modify and extract useful information at runtime, the Fraunhofer CRCG watermark approach makes it possible to detect any modifications that a malicious host tries to perform before or during the execution by embedding an invisible label. A static watermark is embedded into mobile code to detect modifications of code, and a dynamic watermark is embedded to detect runtime modifications of code and data.

In summary, Fraunhofer CRCG has developed complete obfuscation and encrypted execution technologies for stronger mobile code security. Complete obfuscation and encrypted

execution are proactive protection of mobile code against malicious host attacks and the Fraunhofer CRCG Java bytecode watermarking framework provides a passive protection of mobile code by providing means of attack detection. The combination of watermarking, and obfuscation, as well as encrypted execution of Java bytecode, now provides a unified, complementary multi-layer protection – the Fraunhofer CRCG *information armor* for Java-based mobile code.

## 5 References

1. Brickell, E. and Odlyzko, A. Cryptanalysis: A survey of recent results, chapter *Contemporary Cryptology - the Science of Information Integrity*, pages 501–540. IEEE Press, 1992.
2. Collberg, C.; Thomborson, C.; Low, D. (1997). A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997, <http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html>.
3. Cox, I.; Killian, J., Leighton, T. and Shamoon, T. A Secure, Robust Watermark for Multimedia, *Proceedings of Information Hiding First International Workshop*, pp. 185-206, Cambridge, UK, 1996.
4. Dahm, M. (2000). Byte Code Engineering Library, <http://bcel.sourceforge.net/>, 2000
5. Gong, L. (1997). Survivable Mobile Code is Hard to Build. In: *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code Workshop*, 26 - 28 March 1997.
6. Hohl, F. (1998). Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In: *Mobile Agents and Security*. pp 92-113. Springer-Verlag, 1998.
7. Karjoth, G.; Lange, D.; Oshima, M. (1998): A Security Model for Aglets, in: Giovanni Vigna (Ed.): *Mobile Agents and Security*. pp 1-14. Springer-Verlag, 1998.
8. Kassab L.; Voas, J. (1998). Agent Trustworthiness. In: *Proceedings of ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems Secure Internet Mobile Computations*. pp. 121-134, Brussels, July 20-21, 1998.
9. Luo, C.; Zhao, J. (1999): Obfuscating and Watermarking Java Software for Copyright Protection, *Computer Graphik Topics*, No. 4, Vol. 11 (1999), 31-32.
10. Meadows, C. (1997). Detecting Attacks on Mobile Agents. In: *Proc. of the DARPA Workshop on Foundations for Secure Mobile Code Workshop*, 26 - 28 March 1997.
11. Monden, A.; Iida, H.; Matsumoto, K. A Practical Method for Watermarking Java Programs, *Proceedings of the The Twenty-Fourth Annual International Computer Software and Applications Conference*, Taipei, Taiwan, Oct., 2000.

12. Palsberg, J.; Krishnaswamy, S.; Kwon M.; Ma, D.; Shao, Q.; Zhang, Y. Experience with Software Watermarking, *Proceedings of ACSAC'00, 16<sup>th</sup> Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 2000.
13. Sander, T. and Tschudin, C. Protecting Mobile Agents Against Malicious Hosts, in: Giovanni Vigna (Ed.): *Mobile Agents and Security*. pp 44-60. Springer-Verlag, 1998.
14. Sun Microsystems Inc. (1997). Java Card 2.0 Language Subset and Virtual Machine Specification, 1997, Sun Microsystems Inc.
15. Townsend, G. and Collberg, C. Sandmark. <http://www.cs.arizona.edu/sandmark/>
16. Vigna, G. (1998). Cryptographic Traces for Mobile Agents. In: *Mobile Agents and Security*. pp 137-153. Springer-Verlag, 1998.
17. Walsh, T.; Paciorek, N.; Wong, D. (1998). Security and Reliability in Concordia. [http://www.meitca.com/HSL/Projects/Concordia/HICSS98\\_Final.htm](http://www.meitca.com/HSL/Projects/Concordia/HICSS98_Final.htm).
18. Zhao, J. and Koch, E. Embedding Robust Labels Into Images For Copyright Protection, *Proc. of the International Congress on Intellectual Property Rights for Specialized Information, Knowledge and New Technologies*, Vienna, Austria, August 21-25, 1995.
19. Zhao, J. and Luo, C. Digital Watermark Mobile Agents. *Proc. of NISSC'99*, Arlington, Virginia, Oct. 18-21, 1999. 138-146.

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

**BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

☒ **BLACK BORDERS**

☒ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**

☒ **FADED TEXT OR DRAWING**

☒ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**

☒ **SKEWED/SLANTED IMAGES**

☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**

☐ **GRAY SCALE DOCUMENTS**

☒ **LINES OR MARKS ON ORIGINAL DOCUMENT**

☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**

☐ **OTHER:** \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**